Formalizing System F in Coq using Locally Nameless Approach

By Janhavi Doshi for Mechanized Meta-Theory Independent Study.

Abstract : In this project, we have formalized System F in Coq using the Locally Nameless Approach suggested by the paper “Engineering Formal Metatheory.”
(Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. In Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 3–15. POPL ’08. New York, NY, USA: Association for Computing Machinery.)

1. Introduction:

System F, often known as the second-order lambda calculus, is a foundational system in type theory and the study of programming languages. It extends the simply typed lambda calculus by introducing polymorphic types, thereby providing more expressive power. The representation and manipulation of terms and types in type systems are critical for proofs and analyses. There have been many attempts at formalizing system f in Coq using various approaches like locally named representation, using de Bruijn indices, etc. The paper suggests combining locally named representation with de Bruijn representation; this approach is the locally nameless approach. In this report, we investigate the Coq formalization of System F using this approach.

2. Locally Nameless Approach Basics:

The locally nameless method has emerged as a compelling approach to represent variables in formal systems, primarily due to its ability to streamline many of the complexities that arise when handling variables in type systems, lambda calculus, and programming languages.

2.1. De Bruijn Indices

De Bruijn indices serve as a clever and efficient means to represent bound variables. Rather than naming bound variables, each occurrence of a bound variable is replaced by a non-negative integer. This integer represents the 'distance' between the occurrence and its corresponding binder.
Uniformity : Every occurrence of the same bound variable in an expression will have the same index. This uniformity reduces ambiguity and simplifies manipulations.
Alpha-equivalence : Two terms that differ only in the names of their bound variables are naturally treated as identical when using De Bruijn indices. This makes the issue of alpha-conversion trivial, as terms are inherently represented up to alpha-equivalence.

2.2. Atoms

While de Bruijn indices excel at representing bound variables, they're less suited for free variables. That's where atoms come into play. Atoms are distinct entities, typically chosen from an infinite set such as the natural numbers. Each free variable is assigned a unique atom, which stands in for its name.

2.3. Combining De Bruijn Indices and Atoms

In the locally nameless approach, we will use de Bruijn indices to represent bound variables and atoms to represent free variables. By using this mixed representation, many issues related to alpha-conversion (renaming bound variables) are avoided, simplifying certain proofs and manipulations.

3. Basic Definitions

3.1. Atoms

The basic units representing free variables in terms and types are defined as natural numbers. We have defined atoms for terms and types separately to ensure uniqueness and avoid issues related to variable capture.

We have also introduced axioms to guarantee the existence of infinite sets of these atoms and to choose fresh ones not present in given lists.

3.2. Types in System F

There are four kinds of types defined:

Bound type variable (ty_bvar): represented using de Bruijn indices.

Free type variable (ty_fvar): represented using atoms.

Function type (ty_arrow): represents functions from one type to another.

Universal type (ty_forall): represents polymorphism.

3.3. Terms in System F

Six kinds of terms are detailed:

Bound term variable (tm_bvar): like its type counterpart, uses De Bruijn indices.

Free term variable (tm_fvar): uses atoms.

Lambda abstraction (tm_abs): a function with a type annotation.

Application (tm_app): applying one term to another.

Type abstraction (tm_ttabs): a term that's parameterized over a type.

Type application (tm_ttapp): applying a term to a type.

3.4. Contexts

We have defined two contexts, one for variable-type mapping 'type-context', and one for storing type variables 'kind-context'.

4. Well-Formedness Checks

We have implemented closed_ty and closed_tm checks to check for well-formedness of types and terms. Well-formed types and terms are 'locally closed'. We have formalized this through the predicate 't is closed at level k' which is defined recursively.
We have defined 'closed' as 'closed at level 0'.

5. Beta-Reduction and Opening

The paper shows that beta-reduction can be implemented in terms of the substitution function and can also be implemented by generalizing the variable opening operation. Opening with a term u is the same as opening with a fresh variable x and then substituting u for x. We follow this equivalence given by the paper and implement only generalized opening functions in lieu of both opening and substitution. Opening is the process of replacing a de Bruijn index with a free variable. We implement an auxiliary generalized opening helper that replaces a de Bruijn index with a term or a type. By doing this, we get both desired functionalities from the same function and avoid having to implement a separate substitution function. We implement 3 functions for opening types with types, opening terms with types and opening terms with terms.

6. Typing Rules

We have formalized typing rules for system f using Inductive Proposition 'has_type'. We have used our earlier defined 'opening' operations in these rules. Here, we are using these to open types and terms with variables i.e. to replace the de Bruijn index with a free variable.
For our formalization, we are using the 'co-finite' approach. We get a variable 'X' such that it is not in any finite list L. (Recall the axioms we have defined at the beginning). Since we are using cofinitely-quantified typing rules, we do not need to specify what 'x' should be fresh from.
We have also incorporated checks for 'locally closed types' in our typing rules.

7. Values and Reduction Rules

We have defined our 'values' to be locally closed term abstractions and locally closed type abstractions. We will use this inductive proposition in our small step semantics to implement call-by-value.
Our small step semantics are straight-forward. We have defined context rules for term application and type application. Our Beta-Reduction rules for terms checks whether our abstraction term is locally closed and it checks whether our second term is a value. This is similar to how our Beta-reduction rule for types behaves.

8. Conclusion and Future Work

The Coq formalization offers a precise and rigorous description of System F using the locally nameless approach. This representation greatly simplifies the handling of variables, hopefully making proofs and analyses more straightforward. This formalization serves as a robust foundation for further exploration into the properties and proofs regarding System F.

9. References

Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. “Engineering Formal Metatheory.” In Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 3–15. POPL ’08. New York, NY, USA: Association for Computing Machinery.

Charguéraud, Arthur. 2012. “The Locally Nameless Representation.” Journal of Automated Reasoning 49 (3): 363–408.